SPECTRE - MAC - MON

MACRO ASSEMBLER/MONITOR
MICRODRIVE COMPATIBLE
FOR THE SPECTRUM 48K

FROM OASIS SOFTWARE

Author - Philip Harvey

CONTENTS

EDITOR ASSEMBLER
RUN to execute

INTRODUCTION

The SPECTRE-MAC Editor/Assembler is a professional package written for the 48K Sinclair ZX-Spectrum Micro-computer.

The editor gives you the ability to create lines of up to 254 characters. To give you extra long lines the editor uses a unique 'side scrolling' facility. Fast up/down cursor movement can be used, even while in either of the text entry modes. This facility is aimed at speeding up editing. Other facilities incude: find string, replace string, copy block, join files and many more.

With the assembler, macro-instructions can be defined with almost any syntax. Four number bases are available, hex, decimal, binary and octal. Conditional assembly is supported using statements such as IF-END and IF-ELSE-END. Variable loops can be produced using the REPEAT-UNTIL and BREAK commands.

SPECTRE-MAC is a 16K program and occupies the memory from 6000H to A000H, with files at A000H.

Loading
To load SPECTRE-MAC Type: LOAD""

Command Description
A general rule to note is that while the cursor is flashing, you can always abort from any of the commands by pressing CAPS SHIFT/EDIT (except in COMMANDS O, J & W).

<Command A> - Again find

This command will search for the next occurrence of either, the string entered when in command L (LOCATE), or the first string entered while in command R (REPLACE).

The search will begin on the line below the cursor, and will only end if it finds the right string, or it wraps round to reach the end of the cursor line. If the string cannot be found then the words "NOT FOUND" will appear on the status line.

<Command B> - Bottom of file

Once pressed the screen will be instantly updated to show the end section of the present text file. The cursor is set to the top line of text on screen.

<Command C> - Change mode

All other commands are inactive while in CHANGE (and INSERT) mode with the exception of Command A (AGAIN) which can be activated by pressing SYMBOL SHIFT / A.

While in change mode you are not just limited to a screen full of text but have access to the entire file. Cursor keys will still give you fast up/down movement without the need for leaving CHANGE mode.

Tabs can be used in both of the text entry modes. These are entered, for convenience, by pressing SYMBOL SHIFT / SPACE. Tabs are useful in that they format the text with the minimum of memory being used. There are two things you need to remember about tabs:

1. A tab must be fully deleted before it will disappear.

2. Lines are only fully updated when you leave them. Some tabs will be seen to be wrong until you move to another line and the previous line is updated.

A list of special keys is given in fig. 1.0.

<Command D> - Delete lines

Once entered you then have to enter the number of lines (from the cursor line) you wish to erase. There are two exceptions:

1. By entering a '*' instead of a number, you will erase from the cursor line to the end of the file.

2. If there is a block of text you wish to erase but are not sure of its length, go to the end of the block and place on the line just after the block a '-' character. Make sure it is the very first character on the line. Now go to the top of the block, type 'D' followed by '-' and then press ENTER.

<Command G> - Get file

This command is written in basic for convenience. The current file name will be displayed and you will be asked whether you would like to change it. Next you are asked whether you would like to load the file or just verify it. If there is an error in loading you will have to type RUN to re-enter.

<Command H> - Help Mode

If you wish to know which of the command letters does what without having to consult the manual, then press 'H' and any of the command keys. A very brief description of the command will be displayed on the top line.

To exit from this command, type CAPS SHIFT / EDIT.

<Command I> - Insert mode

Same as 'Change mode' (Command C) except that instead of writing over characters, you write in-between them.

<Command J> - Join files

Same as command G (GET) except that the file will be loaded onto the end of the present file.

<Command K> - Kopy block

To move a block of text, simply place delimeters before and after the text, e.g. "-**1" before, and "-" after.

These delimeters must be at the start of each line. Set the cursor to where you want the block to be copied (the cursor MUST NOT be positioned anywhere inside this block). Now simply enter command K and then enter your start delimeter (e.g. -**1) and then press ENTER. While the text is being copied, it will be printed to the screen. The screen may appear disorganised during this operation.

<Command L> - Locate string

It will first ask: Anywhere/start? This refers to where the string is to be found (either at the start of a line or anywhere along the line). It will be set to A-nywhere when you enter, but if you wish to find something only at the start of a line then type 'S' then press ENTER. Now enter the string and press ENTER.

There are two extra facilities available:

1. Entering "\?" will represent any character.
2. Entering "*" will represent any number of characters.

N.B These must not be entered at the beginning or the end of the string because this is ambiguous. If you wish to search for the character "\", then you must enter two of them to represent one ("\\").

<Command O> - Set workspace

If you wish the assembler's workspace to be put after the end of the file, then this command need not be used. To set the workspace address, simply type in the new address and press return. To set workspace back to the end of the file type '*' instead of any address.

<Command P> - Print file

It will first ask whether you are sure, typing 'Y' will output all the characters from the cursor position until it either reaches a line starting with the character '-' or the end of file.

<Command Q> - Quit

Quit will ask you whether you are sure and if you are you will immediately return to BASIC. You can re-enter the editor without losing your file by typing RUN / ENTER.

<Command R> - Replace string

Firstly you will be asked: Stop/Continuous?. A continuous replace would replace a specified number of strings with what ever was required. The STOP mode allows you to see what you would be replacing and gives you a choice of whether to replace or not.

The format is very simple: [Delimeter] First string [Delimeter] Second string [Delimeter] Number to replace.

The delimeter can be any character which does not appear in either of the two strings. All the delimeters must be small characters.

e.g. /STRING TO BE REPLACED/REPLACEMENT STRING/NUMBER TO BE REPLACED

The number has to be entered in decimal and can be anything up to 9999. If you wish to replace all the strings in the file that match, then instead of entering a number, simply enter '*'.

The replace always starts at the cursor line and ends at the end of the file.

<Command S> - Status

Status refers to the left of the top line where you can either display cursor position along line or file end in hex. Simply press the 'S' key until you get the desired status.

<Command T> - Top of file

This will take the cursor and the 'screen window' to the top of the file.

<Command W> - Write file

As with the command G (GET), you have the opportunity of changing the file name before saving.

<Command Z> - Assemble file

After pressing 'Z' you will be asked to enter assembly options (see assembler user guide). Once options are entered press ENTER.

For workspace, see command O.

SYMBOL SHIFT A	Same as command A, but works in CHANGE and INSERT modes.
CAPS SHIFT EDIT	Exit from command.
DELETE	Move back one space, make position blank.
ENTER	This will cause the present cursor line to break in two at the cursor position.
CURSOR LEFT	Move cursor back one character.
CURSOR RIGHT	Move cursor forwards one character.
CURSOR UP	Move cursor up one line. If the key is held down long enough, the cursor will disappear and fast scrolling will be implemented. Cursor will reappear when the key is released (also CURSOR DOWN).
CURSOR DOWN	Move cursor down one line.
CAPS SHIFT SPACE	Clear line from cursor position to end of line.
SYMBOL SHIFT	Insert tab into line.
SYMBOL SHIFT ENTER	Delete to last tab stop or, to last character other than a space.

Fig 1.0 - Table to show command functions which can be used while in CHANGE and INSERT modes.

A	AGAIN	Find next occurrence of previously entered string.
B	BOTTOM	Move to bottom of file.
C	CHANGE	Change text in file.
D	DELETE	Delete a number of lines.
G	GET	Load, or verify, a tape file.
H	HELP	Gives a brief description of the command on the top line.
I	INSERT	Insert into text file.
J	JOIN	Join two tape files together.
K	KOPY	Copy a block of text.
L	LOCATE	Locate a string of text.
O	WORKSPACE	Set new assembler workspace address.
P	PRINT	Send file, from cursor, to printer.
Q	QUIT	Return to BASIC.
R	REPLACE	Replace one string with another string.
S	STATUS	Toggle status.
T	TOP	Display top of file.
W	WRITE	Save file to tape.
Z	ASSEMBLE	Assemble present file.

Fig. 2.0 - This table gives you a brief description of the commands available.

MACRO ASSEMBLER

OVERVIEW

There are many types of macro assembler currently available. Most of these offer a "hardwired" instruction set for a particular machine and allow macros to be defined in terms of these instructions. This is quite reasonable as assemblers are generally used on a single machine to generate code for that machine and a built in instruction set does decrease assembly time. So why depart from this tried and trusted recipe? Well, this assembler was conceived with two main goals in mind: To be able to assemble code for CPU's other than the host. To be able to assemble code in times comparable to assemblers with hardwired instruction sets. In our opinion both these goals were realised.

The assembler comes with a ready compiled library defining the Z80 opcodes (held in RAM for speed in non-disk systems).

Other features supported by this assembler are conditional assembly and even conditional macro statements, iteration directives within macros, access to user defined option flags allowing simple assembly switching without modification of any source code and extensive arithmetic and logical expression handling.

USER GUIDE

This guide assumes a knowledge of normal assembly code.

Code format

Considerable freedom exists in the format of files acceptable to this assembler: Lines can be up to 254 characters long. But there are a few basic conditions that must be adhered to in order that the assembler knows what the code is supposed to mean, these are:

1) Labels/variables may appear anywhere along a line, but if they do not start in column zero then they must be immediately followed by a colon (Unless they are part of/or form an expression).

2) Macro instructions may not appear in column zero as the assembler will confuse them with labels.

3) Files must be terminated by an END statement.

Basic primitives: DB & EQU

It was revealed earlier that the Z80 mnemonic set was not built into the assembler but were defined as macros in terms of more primitive instructions. We will now consider these primitives and define their syntax.

1) Define byte "DB".

This is the most used primitive in instruction set macro definitions (Have a look at the listing of the Z80 instruction macros!). It's job is to generate one byte of code representing the value of the expression or list of expressions following it. These expressions must yield values between -256 and 255 inclusive. Here are some examples of the use and forms of the DB instruction and the code it generates. (This also shows off the expression handling capabilities of this assembler, so see the section on expressions for a detailed desription of these facilities.)

```
0000 00          DB 0
0001 1234        DB 12H,34H            ;A Comment
0003 48454C4C
0007 4F205448
000B 45524521    DB "HELLO THERE!"
000F FE          DB 11111111B - 1
0011 2C          DB "H"-7o<<2
0012 00010203
0016 04050607    DB 0,1,2,3,4,5,6,7
001A FF00FF      DB 1<2 , 5>20, "B"<=42H
  ^          ^   ^
Program      ^   ^
Counter      ^   Code as entered in file
             ^
        Code generated
           by DB
        instructions
```

2) Label equate "EQU"

This statement is the same as in other assemblers and is used to set labels to particular values (which cannot be changed - more on this later). For example:

```
234              LABEL:   EQU 1234H
AAAA             A_NAME:  EQU 1010101010101010B
0C8C             $TT_TT$: EQU 123*64-LABEL1
003F             COLZERO  EQU 77o         ;Label in column zero
^
^
Result of
assignment
is put here
```

Notice that "$"s and "-"s are allowed as part of label names. Labels can be of any length but only the first eight characters are significant.

1) Expressions

As stated earlier; this assembler supports quite extensive arithmetic and logic capabilities. These facilities are no luxury and are used extensively later in macro definitions. Algebraic logic is supported with nestable parentheses and four number bases.

HEX	e.g.	1000H	(base 16)
BINARY	e.g.	10101111B	(base 2)
OCTAL	e.g.	57o	(base 8)
DECIMAL	e.g.	65000	(base 10)

Also here is a list of operators, their functions and precidence.

Operator	Precidence	Use	Function
+	5	+A	Unary Plus
-	5	-A	negate
!	5	!A	Logical complement
>>	4	A>>B	Rotate A right B bits
<<	4	A<<B	Rotate A left B bits
*	4	A*B	Multiply
/	4	A/B	Integer division
%	4	A%B	A modulo B
+	3	A+B	Addition
-	3	A-B	Subtraction
&	2	A&B	Logical AND
	2	A¦B	Logical OR
	2	A~B	Logical XOR
=	1	A=B	Test equality
>	1	A>B	Greater than
>=	1	A>=B	Greater or equal
<	1	A<B	Less than
<=	1	A<=B	Less or equal

Precidence (the relative binding of an operator) is given as a number from 1 to 5 - Where 5 indicates the highest binding power. Of course brackets may be used to alter the order of evaluation of an expression.

Labels and undefined values

In the preceeding section on the EQU instruction it was shown that labels could be set to the value of an expression, and that expression could contain label references. In such situations the reference must be to a previously equated label otherwise a default value of zero will be used. ie:

```
0000            LABEL1: EQU LABEL2*100H
0002            LABEL2: EQU 2
```

With each label there is stored a flag which indicates whether or not the label contains a totally correct value, this will be set true in the second equate but false in the first one. Certain statements that use expressions test this flag and will give an error if the expression contained an uninitialised value. This flag is propagated whenever a label/variable is set to an expression, even through macro arguments (See section on Macros with arguments).

2) Macro definitions

For the uninitiated a macro is a single entity which represents a number of other instructions which can themselves be macros. To illustrate this further let us now define a macro.

```
                DEFMAC ("MACRONAME")   ;This is how we
                                       ;define the macros name

                        DB 77o         ;Generate a byte
                                       ;containing 77 octal
                                       ;This is how we end
                                       ;a macro definition

0000 3F                 MACRONAME      ;This is how we
                END                    ;invoke the macro
                                       ;End of source code
```

Notice that the macro definition did not generate any code but when the macro was invoked it generated a byte containing 77 octal (3F Hexadecimal). So the upshot is that invoking a previously defined macro assembles the code present inside it's macro definition. Macros can be thought of as procedures in some high level languages (Such as PASCAL) which must be defined before their use. This type of macro is of limited use though because it always generates the same code when invoked and in the following sections we will deal with how to define macros which generate different code when invoked in different ways, but it is worth looking at how we can define macros to be used as Z80 instructions with this type of macro:

```
            ;Definitions of the Z80 instructions NOP & HALT
            DEFMAC ("NOP")
                   DB 00H
            END

            DEFMAC ("HALT")
                    DB 76H

            END

0000 00     NOP
0001 76     HALT

            END
```

It is also possible to define macros which consist of more than one word as in this example:

```
              DEFMAC  ("DO-THREE-NOPS")
              DB 00,00,00
              END
0000 000000             DO THREE NOPS
```

Although this is a trivial example it illustrates how "-"s are used to separate the words in multiple word macros. Notice that spaces cannot be used inside the DEFMAC statement.

Macros with arguments

In order that macros can change the code they generate - we must have some method of passing information to macros when they are invoked. We do this by embedding arguments into the macro when we invoke it. There are two types of argument that we need to differentiate between.

1) Numbers, including expressions labels/variables and numeric constants.

2) Constants, including register names and some opcode mnemonics.

Most macros that have register names as arguments usually only use a subset of the total number of registers available and similarly macros that require numeric arguments require only numbers within certain bounds. So to cater for this requirement the assembler has the ability to define sets of registers (Formally called constants) or subranges of numbers.

Constants

Constants are names which can take on a value (a bit like labels) but which are totally distinct from labels and variables in that they cannot be used in expressions. The value a constant takes depends on it's use. Here's how we define constant names:

```
DEFCONST (BC,DE,HL,AF).
```

This definition does not generate any code, of course, and it's purpose is to inform the assembler which names are constant names thus preventing them from being used accidentally as labels.

Sets of constants

We are now in a position to use the constants we defined earlier. To do this we must collect them up into sets; we do this in the following way:

```
DEFSET REG16 = (BC,DE,HL).
```

We have now defined a set called REG16 containing the constants BC,DE and HL. These constants now have a value if used in the context of REG16 - their values are assigned starting at zero and increment for each name in the set, so BC has the value 0, DE has the value 1 and HL has the value 2. It must be emphasised that constants are NOT treated in the same way as labels and cannot be used in expressions and further more; they only have a value when they can be found in a set and their value depends on which set they are used from. To illustrate the use of sets and show how arguments are used we will now consider how we would define a macro which will generate the code for all the Z80 8 bit register to register load instructions. ie: LD B,L etc.

Firstly we must define a set of constants which contains the name of all the registers this instruction is valid for:

```
DEFCONST (A,B,C,D,E,H,L,INVALID).
DEFSET   R8 = (B,C,D,E,H,L,INVALID,A).
```

Notice that the order in which constants are entered into the DEFCONST statement is not important but the order they are put in the DEFSET statement is, and also that a "junk" constant was needed to pad out the list. ie: We need L to have the value 5 and A to have the value 7 and no register has a value corresponding to 6 so a junk name is used.

Now lets define the macro:

```
DEFMAC ("LD*,*",R8,R8)
       DB 40H + #0*8 + #1
END
```

The asterisks in the macro name define where the arguments are to appear and the list of setnames after the macro name string define which set each argument belongs to. Each setname corresponds to each asterisk in the name string. The argument values are passed to the macro in a set of local variables that only the macro may access, these have the names: #n where n has a value in the range 0 to 255.

A look at the opcode for the instruction we are defining shows that it's value is a combination of some preset data and the numbers of the registers it is to use. The DB instruction in the macro will faithfully generate the opcode for this instruction.

Now lets use this macro:

```
0000 78         LD A,B
0001 5F         LD E,A
0002 5A         LD        E  ,  D
```

Notice how spaces are ignored when invoking macros. Spaces are allowed anywhere except in the middle of a name (i.e: HELLO is one name and HE LLO is two names) so spaces can be use to separate arguments from the mnemonic for example.

Number subranges

It is also desirable to define ranges of numbers and attach a name to them as we did with constant sets and so another function of the DEFSET statement is to do just that:

```
DEFSET BYTE = 0 TO 255
```

This has now defined a subrange of numbers called BYTE containing all the numbers from 0 to 255 inclusive. We can now use this set to define another Z80 macro - load immediate 8 bit register ie: LD C,4

```
DEFMAC (\"LD*,*",R8,BYTE)
       DB 6 + #0*8
       DB #1
END.
```

Notice that the only difference between this macro and the last is the type of the second argument, and, so that the assember knows that the macro name string "LD*,*" has already been defined, the "\" is inserted before it in the definition. This must be done whenever a macro name string is repeated otherwise the "Double identifier" error message will be generated.

Here is how we can use this new macro:

```
0003  0620             LD B,00100000B
0005  3EFA             LD A, 0FAH
0007  2645             LD H,"E"
```

It is worth pointing out that constants may be used in any number of different sets and the user must make sure that ambiguities do not arise because of this.

Certain Z80 instructions require an argument that has only one value to follow them (Such as: EX DE,HL are constant). Using the above methods of defining macros we would have to define two sets: one with HL in an the other with DE in. Which is not desirable. So in order to obviate the need to do this constant names may be used in macro definitions and will mean a set with one element. To illustrate this let us define the macro EX DE,HL:

```
;Assuming that HL and DE have been defined as constants

                 DEFMAC ("EX*,*",DE,HL)
                        DB 0EBH
                 END.
0000 EB                 EX DE,HL          ;Use of this macro
```

By now you may be wondering why we did not define a macro that used no arguments in the above case. Such as:

```
DEFMAC ("EX-DE,HL")            ;This is illegal
       DB 0EBH
END.
```

If the names DE and HL were not defined as constants this macro would be perfectly acceptable but as they have been defined as constants the assembler would search for a macro mask like this: "EX*,*" and so the "Undefined" error will occur.

With the facilities so far covered it should now be possible for the user to write his/her own macros which can simulate any mnemonic in the Z80 instruction set and much more.

Conditional assembly

The main conditional assembly statement supported by this assemler is the IF-END and the IF-ELSE-END statement combinations. These are used to enclose parts of the assembly code and enable/disable the assembler assembling it. This is mostly used for allowing once source listing to produce a number of different versions of a program depending on 'switches' set at the start of the source code. Here is a brief example of it's use:

Here is a notional data storage routine; for disk systems it must contain code to write to the disks and for tape systems it must contain code to write to tape:

```
                 DISK:   EQU 0
                 TAPE:   EQU 1

                 SWITCH: EQU DISK

                 ...Lots of source code...
      WRITE:
                 IF SWITCH=TAPE

                 ...Tape interface code...

                 END                         ;*

                 IF SWITCH=DISK              ;*

                 ...Disk interface code...

                 END

                 ...Rest of source file...
```

* - The code so indicated can be replaced with a single ELSE statement.

One point worthy of note is that the IF statement works by evaluating the expression and if it is zero it is taken as 'false', if it is non-zero it is considered 'true'. This means that the first IF statement could have been replaced with the following:

```
                 IF SWITCH
```

But in this example it makes the statement less clear.

The IF statement may also be used inside macro definitions, enabling macros to change at invocation time the way they assemble.

Here is a small but useful example to illustrate this:

Supposing we wish to define a macro which performs the following operation: LD rr,(HL) where rr is another 16 bit register from the set HL,DE,BC. so if we were to invoke LD BC, (HL), the following instructions would be used:

```
1)      LD C,(HL)       ;Get low byte
2)      INC HL
3)      LD B,(HL)       ;Get high byte
4)      DEC HL          ;Restore HL
```

A similar sequence would be required for the register DE.

On the other hand we might wish to do this instruction: LD HL,(HL). In this case the following instructions would be needed:

```
1)      PUSH AF         ;Save AF
2)      LD A,(HL)       ;Get low byte
3)      INC HL
4)      LD H,(HL)       ;Get high byte
5)      LD L,A          ;Load up low byte
6)      POP AF          ;Restore AF
```

So we need three different code sequences for each type of instruction. This can be done with the IF statement in a macro definition:

```
                 DEFCONST (HL,DE,BC).

                 DEFSET RR = (BC,DE,HL).

                 DEFMAC ("LD*,(*)",RR,HL)

                 IF #0=0                 ;Case for LD BC,(HL)
                        LD C,(HL)
                        INC HL
                        LD B,(HL)
                        DEC HL
```

```
                    ELSE IF #0=1             ;Case for LD DE,(HL)
                            LD E,(HL)
                            INC HL
                            LD D,(HL)
                            DEC HL
                    ELSE IF #0=2             ;Case for LD HL,(HL)
                            PUSH AF
                            LD A,(HL)
                            INC HL
                            LD H,(HL)
                            LD L,A
                            POP AF
                    END END END              ;One for each IF
                    END

0000  F57E2366
0004  6FF1                  LD HL,(HL)       ;Invocations of the
                                             macro
0006  4E23462B              LD BC,(HL)
000A  5E23562B              LD DE,(HL)
```

Argument modification

When using macros inside macros it is possible to modify the value that a macro definition sees a constant argument as. This is done by immediately following the argument in the macro invocation with a value to be added on to the constants (set related) value in square brackets: ie: LD C 1 ,A if invoked inside a macro would generate the code for LD D,A (See the Z80 macro library).

This facility can be used to simplify the above example:

```
                    DEFMAC ("LD*,(*)",RR,HL)
                    IF #0<2                         ;BC & DE case
                            LD C #0*2 ,(HL)
                            INC HL
                            LD B #0*2 , (HL)
                            DEC HL
                    ELSE
                            PUSH AF
                            LD A,(HL)
                            INC HL
                            LD H,(HL)
                            LD L,A
                            POP AF
                    END
                    END.
```

So although this is a complex facility to use it does simplify the code in certain circumstances.

Iteration and the variable

It is often useful for macros to produce tables etc, and to do this they must be able to have a conditional looping statement - This assembler has two: The $WHILE - END sequence and the $REPEAT - $UNTIL sequence and there is a $BREAK statement for breaking out of these sequences. Their use is best illustrated by example because they are not very different from similar statements in high level languages.

Here is a macro which will fill a number of bytes with a particular value at assembly time:

```
                    DEFVAR (COUNTER).

                    DEFSET NN = 0 TO 256.

                    DEFMAC ("FILL*BYTES-WITH*",NN,NN)
                    COUNTER:= #0

                    $WHILE COUNTER>0
                            DB #1
                            COUNTER:=COUNTER-1
                    END
                    END.

0000 AAAAAAAA
0004 AAAAAAAA
0008 AA                     FILL 9 BYTES WITH 0AAH ;Heres how we use it.
```

Notice that we used what looked like a label as a counter. This was in fact a variable and was defined as such in the DEFVAR statement which works in much the same way as DEFCONST. Variables may be used anywhere that labels are used but must be assigned to using the "=" rather than the EQU for normal labels. The advantage of variables over labels is that variables may be assigned to more than once without the assembler objecting. They can, of course, be used anywhere labels are used and can sometimes be used to advantage instead of labels but it is best to restrict their use, otherwise mistakes can be made which will not be picked up by the assembler.

The $REPEAT is used in much the same way as the $WHILE except that the condition is tested at the end of the loop rather than at the beginning.

```
ie:                 $REPEAT

                            ...some code...

                    $UNTIL expression
```

When the $BREAK statement is encountered inside one of the above loops, assembly breaks from it's current position and continues after the end of the loop. This can also be used to break out of macros.

Assembler terminating information

When the assembler terminates it prints on the screen a few statistics about the assembly: The final value of the pseudo program counter ($), the final value of the LOAD pointer, the start and end addresses of the workspace used and if any errors- the number of errors and the assembler pass they occurred on.

Creating a library

As has been stated earlier this assembler comes with a built in macro library containing all the Z80 instruction set. It is possible for the user to change this library and thus personalise the assembler as little or as much as he likes! It would be possible, for instance, to get rid of the Z80 set and put in the macro definitions for the 6502 instruction set. One need not even put in micro instruction sets; the user could create his own language made out of macros.

To add a new library the user must first write it as he would any other assembler program (The Z80 macro library built into the assembler is supplied on the tape provided so that the user may modify it's contents). The file is then assembled using the "K" option. The tables thus generated are then put at the end of the assembler at 9175H in front of the source file (modify address at 6148H to change file start address and reload file). The version supplied has the source file starting at 9FF0H and the Z80 macro library comes to about 9FC6H so little room is left. It is worth leaving as much room as possible for the new library by moving the file start and then moving the file start down to just past the end of the library when finished. The end of the new library is indicated by the second field of the workspace section in the assembler terminating information. (When using the "K" option the first field will point to the end of the file as usual). This "new" assembler may then be saved on tape (6000H to end+1 of workspace).

GENERAL USER GUIDE

Assuming that a file has been created, containing assembly source code in the correct format, it is assembled as follows:

1) Type 'Z' while in the editor.
2) Enter option letters required.
3) Press ENTER and wait.
4) Either, assembler will return with no errors or any errors will be indicated and the user must correct and reassemble.

Guide to options

E - Stop and indicate error position with cursor.
L - Produce listfile.
S - Produce symbol table of labels/variables.
T - Produce other symbol tables (macros, sets & constants).
P - Send any output to printer.
M - If file contains a load directive then store object in memory.
K - Create a new "built in library" (See section on libraries.)

Load directive

This will cause any object generated to be put into memory at address following the load address. (If the 'M' option is on)

i.e.:-

```
                    ORG 1000H
                    LOAD 8000H

1000 47             LD B,A
1001 21 02 00       LD HL,2

                    END
```

will put the hex codes 47,21,02 & 00, starting at 8000H, into ram (Although the code is assembled as if it was to go at 1000H).

LIST directive

This is a macro in the built in library which will switch on an off the list option (above) whilst printing a source listing It is used as follows:

```
        ....source code (A)

        LIST      OFF

        ....source code (B)

        LIST      ON

        ....source code (C)
```

In this example (providing the list option was used in invoking the assembler) section (A) would be listed, section (B) would be missed and section (C) would be listed. If the assembler was not invoked with the list option then only section (C) would be listed.

System variables

Certain system variables are available for use (with care) by the source file during assembly. These are accessed by putting"$n" where n is between 0 & 9. These may be used anywhere a standard label/variable is used and contain the following:-

$0 (or just $)	Pseudo program counter.
$1	Load address pointer.
$2	Option flags.

-- bit 0	reserved
bit 1	print flag
bit 2	reserved
bit 3	list option
bit 4	symbol option
bit 5	reserved
bit 6	reserved
bit 7	error stop option
bit 8	'T' option
bit 9	'M' option
bit 10	Store to memory enable
bit 11	reserved
bit 12	'0' option
bit 13	'1' option
bit 14	'2' option
bit 15	'3' option

The '0', '1', '2' & '3' options are not used by the assembler but are available as a method of externally passing information into an assembling source file. This could be used in conjunction with the conditional IF statements to perform assembly switches without affecting the source file. ie:

```
IF $2 & 0001000000000000B

...source code (A)

ELSE

...source code (B)

END
```

So, if the assembler was invoked with the "O" option source code (A) only would be assembled and if the "O" option was omitted then only source code (B) would be assembled.

Errors

There are ten different error responses and each of these will cover a number of different but similar error conditions.

Here is a list of their meanings:

PHASE -- A label had a different value on pass three to that on pass two. This can be caused by macros generating different code on each pass, or labels not being initialised correctly.

SIZE -- This encompasses a number of conditions mostly concerned with the size of a value.

SYMBOL -- A symbol of one type was found where one of another type was expected.

ARGM -- Argument error. Usually on macro definitions.

UNDEF -- A symbol/macro has not been defined.

SYNTAX -- Syntax error often occurs in macro name strings when illegal characters are used.

INFINITE -- A loop using $WHILE or $REPEAT has iterated over 65535 times and is possibly an infinite loop.

INITIAL -- Label initialisation error.

EOF -- Unexpected end of file found. Usually too few END's in file.

SYS -- System error - This is generated if a library was made which overran the start of the source file.

SPECTRUM MONITOR (SPECTREMON)

LOADING
SPECTREMON occupies the memory from F000H to FFFFH (7000H to 7FFFH on 16k machines). There are two copies supplied:

Side 1 - 48k version
Side 2 - 16k version

SPECTREMON has a small BASIC loader which makes sure the stack is below the program.

```
LOAD""
```

Now load your own program (to be debugged) if you have not already done so. Type:

RANDOMIZE USR 61440 (28672 on 16k machines) or RUN 9999 to execute SPECTREMON.

INTRODUCTION

Machine code is very much harder to debug than BASIC since are no error messages to help you. Usually what happens is the system crashes and it is impossible to tell where the fault was. Some faults can be found just by following through your source code, but others are very difficult to find. This is where SPECTREMON comes in. SPECTREMON allows you to follow though your programs, one instruction at a time, with the help of a 'Front panel' display. The front panel occupies the top nine lines of the screen, while active, and does not scroll. All the active registers are displayed (except the 'R' register which is constantly changing) plus the memory to which they point.

The FRONT PANEL Explained.

```
>PC 0000 DI
 SP EFE8 ( 2D2B )  I 3F
 IX 0000 F3 (IX+00H) F3 11110011
 IY 5C3A FF (IY+00H) FF 11111111
 AF 0054 Flags  Z H P   00000000
 HL 2D2B FD 21  (HL) FD 11111101
 DE 5CF9 00 00 00 F0 00 20 20 20
 BC F000 ED 73 D8 FF CD 65 F3 CD
```

In the top left hand corner, there is an arrow. This arrow is used for changing the values of the 16-bit registers down the left of the display. The arrow is moved using the up/down cursor keys and the 'R' command is used to change their values.

The first line shows the value of the PC (Program counter) plus the mnemonic at that address.

The second line shows the value of the SP (Stack Pointer) followed by the last 16-bit number to be placed on the stack. Along the same line is the value of the 'I' register.

The third line shows the value of the 'IX' (Index Register) followed by the value at that address. Next along the line the current (IX+dd) followed by the value at this position, in hex and then in binary.

The fourth line is identical to the last except the register in question is now the IY (Index Register).

The fifth line shows the value of the HL register pair followed by the value at this address and the one after. Next, the value at the HL address is shown again along with it's binary equivalent.

The sixth line shows the value of the DE register pair followed by the value at that position and the seven positions after that.

The seventh line is identical to the last except that BC is now the register pair being displayed.

Commands in Detail

All values displayed are in hex with the exception of a few in binary. All values entered must also be in hex and not decimal.

<B> - Breakpoint

You can put SPECTREMON into a continuous mode using the 'SO' command. There are two ways to stop the continuous mode:

1. Press the SPACE key.
2. When a breakpoint is encountered.

There are ten breakpoints that can be individually set to any address. Each breakpoint has it's own down counter. What this means is that if, for example, you set the down counter to 5, SPECTREMON would stop the continuous mode after it has encountered this breakpoint five times.

These breakpoints are only activated while in 'SO' mode (see S command).

a) Set a breakpoint
   Format: Bbb aaaa cccc

Breakpoints can be set at the beginning of any Z80 instruction, even if the code is in 'Read Only Memory' (ROM). To set the breakpoint enter the command letter (B) followed by the breakpoint number (bb). This can be anything between 0 and 9. Next enter the address at which you want the breakpoint to be placed (aaaa) followed by a down counter (cccc). The breakpoint will only be activated when cccc reaches zero. If you want a breakpoint to be activated the first time it is found then set cccc to 1. If, for example, you want the breakpoint activated after 4096 times round, then set cccc to 1000 (hex). When a counter reaches zero it is then reset to is initial value. After pressing 'ENTER' all the active breakpints will be displayed in the same format as they are entered.

b) Reset a breakpoint.
   Format: Bbb

To deactivate a breakpoint, just enter the breakpoint number (bb). All the active breakpoints will then be displayed.

c) Display all active breakpoints.
   Format: B

This will display all the active breakpoints in the following format:

```
01  3F45  0001
!   !     Down counter (cccc)
!   !
!   Breakpoint address (aaaa)
!
Breakpoint number (bb)
```

<C> - Copy
      Format: Cvv ssss llll

This command will fill an area of memory with a specified value (vv) starting at address (ssss) and of length (llll).

Example:

```
)CFF 4000 1800          will fill the screen with FFH's.
```

<D> - Disassemble
      Format: Dssss eeee llll

This will disassemble the instructions from address (ssss) and finish at address (eeee). The command will pause after (llll) lines have been displayed. Press any key except 'SPACE' to continue. Abort command by pressing 'SPACE' even if it is still scrolling. If (llll) is not entered the value will default to the number of lines available for scrolling.

Example:

```
)D0000 0010 0000          will display the mnemonics of the
                          instructions between addresses
                          0000H and 0010H.

0000 DI
0001 XOR A
0002 LD DE,FFFFH
0005 JP 11CBH
0008 LD HL,(5C5DH)
000B LD (5C5FH),HL
000E JR 0053H
```

      Format: D

This will disassemble from the current PC (PROGRAM COUNTER) address. Press 'SPACE' to abort command.

<E> - Execute
      Format: Essss

This will execute the program at address (ssss). A return from any program will take you back to SPECTREMON provided you do not corrupt the stack.

Example:

```
)M6000
6000 (00) C9.
)E6000
```

RANDOMIZE USR 61440 (28672 for 16k) will return you to SPECTREMON.

Format: E

This will execute the program at the current PC address.

<F> - Frontpanel
Format: F

This will either activate the frontpanel or deactivate it depending on its current status. Use in conjunction with the S (Single Step) command. The frontpanel is on when you first enter SPECTREMON.

<I> - Intelligent copy
Format: Issss dddd llll

This will copy the data, of length (llll), from address (ssss) to address (dddd). This is an intelligent copy so even if these two blocks overlap the data will not be corrupted.

Example:

)I7000 6FFF 1000 will copy 1000H bytes from 7000H to 6FFFH without corrupting it.

<J> - Jump
Format: Jaaaa

This sets a breakpoint at address (aaaa) and executes the code at the present PC address. This is different from the normal breakpoints in two ways:

1. This cannot be used inside ROM.
2. This is quicker than using the normal breakpoints as this does not step through the instructions.

Format: J

This will set a breakpoint immediately after the instruction at the PC address. This is useful, for example, if your program calls a routine in the spectrum monitor. When the PC points to this CALL you can simply use the 'J' command to execute the routine and stop it once it returns back to your program.

<M> - Memory
Format: Maaaa

This command allows you to modify any part of memory. (aaaa) is the start address. Enter values along the current line and then press 'ENTER' to enter them into memory. There are also other symbols you can use in this mode.

: Move back one byte.
, Enter Ascii (e.g. ,R = 52H)
/ Change address (e.g. /1000 will take you to address 1000H).
. Exit from command.

Example:

```
)M6000
6000 (C9)  21 00 60 3E FF ED B1
6007 (00)  C9/7000
7000 (00)  ,H,E,L,L,0 FF
7006 (00)  :
7005 (FF)  :
7004 (4F)  .
```

<O> - Output
Format: Opppp vv

Output value (vv) to Port (pppp).

Example:

```
)OFE 00 will output 00 to the port 00FE (border port).
)
)OFE FF
```

<P> - Printer
Format: P

Toggle printer on/off. The frontpanel will always be sent to the printer after any command, while it is active.

<Q> - Query
Format: Qpppp

Input from port (pppp) and display value.

Example:

```
)Q0 will input from port 0000.
FE
)Q
```

<R> - Register
Format: Rvvvv

With the frontpanel active there is an arrow at the top left of the screen. This points to a register. This command will alter the register value, pointed to by the arrow, to (vvvv). You can use the cursor keys, at any time while the cursor is flashing, to move this arrow to point to the desired register.

Example:

If the arrow is at the top and you want to single step through a program at 6000H then use the 'R' command to set up the PC address:

)R6000 will set PC to 6000H.

<S> - Step
Format: S

Step through one instruction at the current PC address. Use the 'R' command to initially set the PC value.

Format: S0

This is the same as 'S' on it's own except this will continue stepping until either: a breakpoint is encountered, or you press the 'SPACE' key. This can be used even if the frontpanel is not on the screen and will be considerably faster.

<T> - Tabulate
Format: Tssss eeee llll

This will tabulate the memory (in hex) from address (ssss) to address (eeee) stopping after (llll) lines have been displayed. If (llll) is not entered, the value will default to the number of lines available.

Example:

```
)T0000 0020 1
0000 F3 AF 11 FF FF C3 CB 11 (wait for keypress)
0008 2A 5D 5C 22 5F 5C 18 43 (keypress)
0010 C3 F2 15 FF FF FF FF FF
0018 2A 5D 5C 7E CD 7D 00 D0
```

Format: Tssss eeee llll 0

Same as above except this will also display the Ascii values.

Example:

```
)T0100 0110 0 0
0100 4F D2 41 4E C4 3C BD 3E
     O  .  A  N  .  <  .  >
0108 BD 3C BE 4C 49 4E C5 54
     .  <  .  L  I  N  .  T
```

Notice that any characters below 20H and characters above 80H are represented by a dot.

Format: T

This will tabulate from the current PC address pausing after filling the screen. Press any key to continue except 'SPACE' which will abort the command even while tabulation is still taking place.

<X> - Xchange
Format: X

This will allow you to display the alternate register set. If you are single stepping a program remember to swop the registers back again before proceeding.

<Y> - Clear screen
Format: Y

Clear screen and display frontpanel (if active).

<Z> - BASIC
Format: Z

Return control back to the BASIC interpreter.

COMMAND SUMMARY

Command	Arguments	Command description

B	bb aaaa cccc	SET A 'SOFT' BREAKPOINT
		Where: bb = Breakpoint number (0 to 9)
		aaaa = Breakpoint address
		cccc = Down counter

B	bb	RESET A 'SOFT' BREAKPOINT Where: bb = Breakpoint number
B		DISPLAY 'SOFT' BREAKPOINT VALUES
C	vv ssss llll	COPY VALUE THROUGH MEMORY Where: vv = Value to copy through memory ssss = Address at which to start llll = Length of copy
D	ssss eeee llll	DISASSEMBLE INSTRUCTIONS Where: ssss = Start address eeee = End address llll = Number of lines before pausing
D		DISASSEMBLE INSTRUCTIONS STARTING AT PC ADDRESS
E	ssss	EXECUTE PROGRAM Where: ssss = Start address
E		EXECUTE PROGRAM AT THE PC ADDRESS
F		TOGGLE FRONTPANEL ON/OFF

I	ssss dddd llll	INTELLIGENT COPY
		Where: ssss = Source address
		dddd = Destination address
		llll = Length of copy

J	aaaa	SET 'HARD' BREAKPOINT Where: aaaa = Breakpoint address
J		SET 'HARD' BREAKPOINT JUST AFTER THE PRESENT INSTRUCTION
M	aaaa	MODIFY MEMORY Where: aaaa = Start address
M		MODIFY MEMORY AT THE CURRENT PC ADDRESS
O	pppp vv	OUTPUT A VALUE TO A PORT Where: pppp = Port number vv = Value to output
P		TOGGLE PRINTER OUTPUT ON/OFF

Q	pppp	QUERY PORT Where: pppp = Port number
R	vvvv	CHANGE REGISTER VALUE Where: vvvv = New register value
S		STEP THROUGH ONE INSTRUCTION
SO		STEP THROUGH INSTRUCTIONS WITHOUT STOPPING
T	ssss eeee 1111	TABULATE MEMORY Where: ssss = Start address eeee = End address 1111 = Number of lines before pausing
T	ssss eeee 1111 0	TABULATE MEMORY PLUS ASCII Where: ssss = Start address eeee = End address 1111 = Number of lines before pausing
T		TABULATE FROM PC ADDRESS
X		TOGGLE BETWEEN REGISTERS AND ALTERNATE REGISTERS
Y		CLEAR SCREEN
Z		RETURN TO BASIC

MICRODRIVES

A version of the SPECTRE-MAC-MON for use with microdrives is saved after the standard tape version. Essentially it is identical to the tape version but relocated up by 1k, i.e. 1024 decimal which is 400H.

To save this onto microdrive use the following procedure:

1. Type CLEAR 25599.
2. Place the SPECTRE-MAC (microdrive version) cassette into the recorder (SPECTRE-MAC side up).
3. Type LOAD"":LOAD"" CODE.
4. Now save to microdrive cartridge by typing

```
SAVE*"m";1;"SPECTREMAC" LINE 9999
SAVE*"m";1;"RC" CODE 25600,16384
```

To load SPECTRE-MAC from cartridge simply type:

```
LOAD*"m";1;"SPECTREMAC"
```

To save the monitor to microdrive use:

1. CLEAR 28671 (16k version) CLEAR 61439 (48k version)
2. Place the SPECTRE-MON (16k or 48k version) cassette into the recorder.
3. Type MERGE"":LOAD"" CODE and wait for both parts to load.
4. Now save to microdrive cartridge by typing

```
SAVE*"m";1;"SPECTREMON"LINE 9999
SAVE*"m";1;"MON" CODE28672,16384 (16k version), or
SAVE*"m";1;"MON" CODE61440,16384 (48k version).
```

To load from microdrive just type

```
LOAD*"M";1;"SPECTREMON"
```

To save the MACRO library to microdrive:

1. After loading SPECTREMAC from microdrive use the 'Q' command to return back to BASIC.
2. Type LOAD"MACROS" CODE 41984
   This will load the macro library from tape.

Once loaded, type RUN to re-enter the editor and save the macro library using the 'W' command.

Finally, there are references in the manual to addresses for the tape version. The microdrive versions are all 1024 (400k) greater. The addresses should be changed as below.

Page 1, paragraph 4	- 6000H becomes 6400H.
	A000H becomes A400H.
Page 16, paragraph 1	- 9175H becomes 9575H.
	6148H becomes 6548H.
	9FFDH becomes A3FDH.
	9FC6H becomes A3C6H.
	6000H becomes 6400H.

SPECTRE - MAC

Spectre-Mac is simply the most comprehensive Assembler available for the Spectrum 48K. The essential difference between Spectre-Mac and the rest is the facility to define macros. The manual explains how this powerful facility can be used to assemble code for CPU's other than the Z80.

Some of it's many features include.

* Full Screen editor with control commands to locate and change strings, copy blanks, insert strings, delete strings and much more. The Editor even has a unique side scroll facility so that lines of up to 254 characters can be entered. The ZX-Printer is also fully supported.
* All the standard primitives and directives such as DB, EQU, LOAD, ORG etc.
* Extensive arithmetic and logical capabilities. Expressions can contain nestable parenthesis and four number bases can be used. There are no less than 18 arithmetic and logical operators.
* MACRO DEFINITIONS

```
DEFMAC (X, Y,...)
DEFCONST (X, Y,...)
DEFSET VARS = (X, Y, Z ...)
DEFVAR (X)
```

* Macros can be used within macros.
* Conditional Assembly
  IF .. END IF .. ELSE ... END
* Iteration and variables $WHILE - END, $REPEAT - $UNTIL
  $BREAK
* Complete macro library supplied (The Z80 instruction set). User can personalise this as much as he likes. For instance, remove the Z80 instruction set and replace it with the 6502 instruction set.
* Comprehensive list of error reports.
* Concise easy to follow manual.

SPECTRE - MON

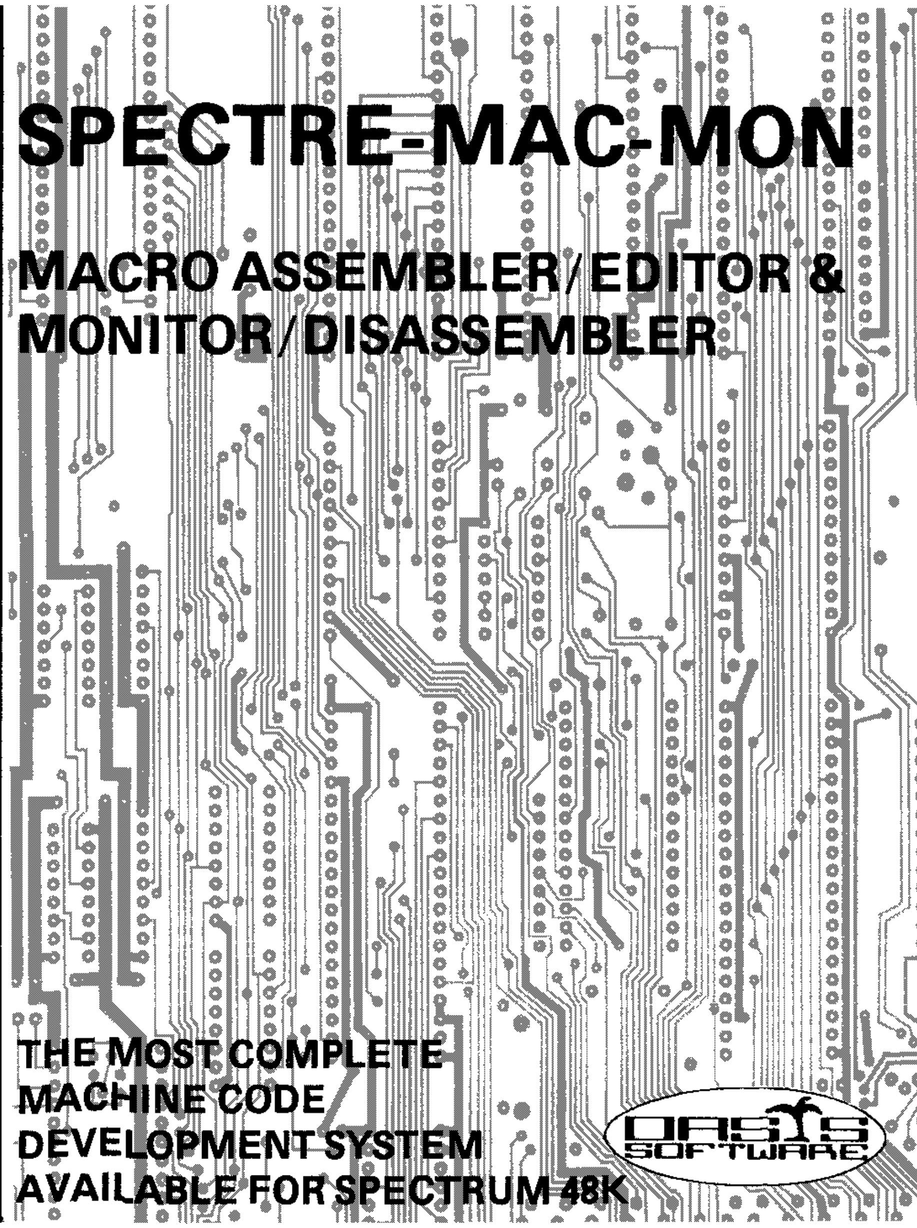
Used in conjunction with Spectre-Mac, Spectre-Mon provides fast simple machine code program development. It's many powerful features include.

* Up to 10 breakpoints can be set in RAM or ROM each with a counter which only activates the breakpoint at the predetermined occurrence. Breakpoints can also be displayed or reset at will.
* Copy value through memory.
* Disassemble RAM/ROM.
* Optional "frontpanel" which displays register contents, the contents of the address held in the registers etc.
* Modify register or memory.
* Intelligent copy to enable blocks of code to be moved without overwriting itself.
* Direct Input/Output to ports.
* Single step execution with or without 'frontpanel' display.
* Display memory and if required it's ASCII representation.

OASIS SOFTWARE
Alexandra Parade
Weston-super-Mare
Avon BS23 1QT
Tel. 0934 419921

Any Oasis product failing to Load will be replaced without question.

SPECTRE-MAC-MON (48K)



SPECTRE-MAC-MON

MACRO ASSEMBLER/EDITOR & MONITOR/DISASSEMBLER

THE MOST COMPLETE
MACHINE CODE
DEVELOPMENT SYSTEM
AVAILABLE FOR SPECTRUM 48K

OASIS SOFTWARE

SPECTRE - MAC

Spectre-Mac is simply the most comprehensive Assembler available for the Spectrum 48K. The essential difference between Spectre-Mac and the rest is the facility to define macros. The manual explains how this powerful facility can be used to assemble code for CPU's other than the Z80.

Some of it's many features include.

* Full Screen editor with control commands to locate and change strings, copy blanks, insert strings, delete strings and much more. The Editor even has a unique side scroll facility so that lines of up to 254 characters can be entered. The ZX-Printer is also fully supported.
* All the standard primitives and directives such as DB, EQU, LOAD, ORG etc.
* Extensive arithmetic and logical capabilities. Expressions can contain nestable parenthesis and four number bases can be used. There are no less than 18 arithmetic and logical operators.
* MACRO DEFINITIONS

  DEFMAC (X, Y,...)
  DEFCONST (X, Y,...)
  DEFSET VARS = (X, Y, Z ...)
  DEFVAR (X)
* Macros can be used within macros.
* Conditional Assembly
  IF .. END IF .. ELSE ... END
* Iteration and variables $WHILE - END, $REPEAT - $UNTIL
  $BREAK
* Complete macro library supplied (The Z80 instruction set). User can personalise this as much as he likes. For instance, remove the Z80 instruction set and replace it with the 6502 instruction set.
* Comprehensive list of error reports.
* Concise easy to follow manual.

SPECTRE - MON

Used in conjunction with Spectre-Mac, Spectre-Mon provides fast simple machine code program development. It's many powerful features include.

* Up to 10 breakpoints can be set in RAM or ROM each with a counter which only activates the breakpoint at the pre-determined occurrence. Breakpoints can also be displayed or reset at will.
* Copy value through memory.
* Disassemble RAM/ROM.
* Optional "frontpanel" which displays register contents, the contents of the address held in the registers etc.
* Modify register or memory.
* Intelligent copy to enable blocks of code to be moved without overwriting itself.
* Direct Input/Output to ports.
* Single step execution with or without 'frontpanel' display.
* Display memory and if required it's ASCII representation.

OASIS SOFTWARE
Alexandra Parade
Weston-super-Mare
Avon BS23 1QT
Tel. 0934 419921

Any Oasis product failing to Load will be replaced without question.

SPECTRE-MAC-MON (48K)

SPECTRE-MAC-MON

MACRO ASSEMBLER/EDITOR & MONITOR/DISASSEMBLER

THE MOST COMPLETE MACHINE CODE DEVELOPMENT SYSTEM AVAILABLE FOR SPECTRUM 48K

OASIS SOFTWARE